Conozca la API de fechas de Java 8

Alexandre Aquiles
27/11/2020

Manipular fechas en Java siempre ha sido una tarea ardua. En Java 1.0 solo existía la clase Date, que era complicada de usar y no funcionaba bien con la internacionalización. Con el lanzamiento de Java 1.1, surgió la clase abstracta `Calendar`, con muchas más funciones, pero con mutabilidad y decisiones de diseño cuestionables.

Comenzando con Java 8, una nueva API de fechas está disponible en el paquete java.time. Esta API es una excelente adición a las bibliotecas estándar de Java.

Esta API se basó en la famosa biblioteca JodaTime, que fue el sustento para lidiar con las fechas hasta entonces. La nueva API no es exactamente la misma que JodaTime, ya que varios detalles conceptuales y de implementación han sido mejorados.

Uno de los conceptos principales de esta nueva API es la separación de cómo datos temporales se interpretan en dos categorías: la de las computadoras y la de los humanos.

Fechas para computadoras

Para una computadora, el tiempo es un número que crece a todo momento. En Java, históricamente se usó un long que representaba los milisegundos desde el 01/01/1970 hasta las 00:00:00. En la nueva API, la clase `Instant`, se utiliza para representar ese número, ahora con precisión de nanosegundos.

```
Instant ahora = Instant.now();
System.out.println(ahora); //2020-11-08T10:02:52.036Z (formato ISO-8601)
```

Podemos usar un `Instant`, por ejemplo, para medir el tiempo de ejecución de un algoritmo.

```
Instant inicio = Instant.now();
rodaAlgoritmo();
Instant fin = Instant.now();

Duration duracion = Duration.between(inicio, fin);
long duracionEnMilisegundos = duracion.toMillis();
```

Observe que usamos la clase `Duration`. Esta clase se usa para medir una cantidad de tiempo en términos de nanosegundos. Puede obtener esta cantidad de tiempo en varias unidades llamando a métodos como `toNanos`, `toMillis`, `getSeconds`, etc.

Fechas para humanos

Para un ser humano, por otro lado, existe una división del tiempo en años, meses, días, semanas, horas, minutos, segundos, etc. También tenemos zonas horarias, horario de verano y diferentes calendarios.

Surgen varias preguntas al considerar la interpretación humana del tiempo. Por ejemplo, en el calendario judío, un año puede ser de 13 meses. Las clases de paquetes `java.time` permiten que estas interpretaciones del tiempo se definan y manipulen con precisión, al contrario de lo que sucedió al usar `Date o Calendar`.

Tenemos, por ejemplo, la clase `LocalDate` que representa una fecha, es decir, un período de 24 horas con un día, mes y año definidos.

```
LocalDate hoy = LocalDate.now();
System.out.println(hoy); //2014-04-08 (formato ISO-8601)
```

Un `LocalDate` sirve para representar, por ejemplo, la fecha de emisión de nuestra CI, en el que no nos importan las horas ni los minutos, sino todo el día. Podemos crear un `LocalDate` para una fecha específica usando el método `of`:

```
LocalDate emisionCI = LocalDate.of(2000, 1, 15);
```

Tenga en cuenta que usamos el valor 1 para representar el mes de enero. Podríamos haber usado el enum `Month` con el valor *JANUARY*. Todavía existe el enum `DayOfWeek`, que representa los días de la semana.

Para calcular la duración entre dos `LocalDate`, debemos usar un `Period`, que ya se ocupa de los años bisiestos y otros detalles.

```
LocalDate hombreEnElEspacio = LocalDate.of(1961, Month.APRIL, 12);
LocalDate hombreEnLaLuna = LocalDate.of(1969, Month.MAY, 25);

Period periodo = Period.between(hombreEnElEspacio, hombreEnLaLuna);

System.out.printf("%s años, %s mes y %s días",
    periodo.getYears() , periodo.getMonths(), periodo.getDays());
    //8 años, 1 mes y 13 días
```

La clase `LocalTime` sirve para representar solo una vez, sin fecha específica. Podemos, por ejemplo, utilizarlo para representar el momento de entrada al trabajo.

```
LocalTime horarioDeEntrada = LocalTime.of(9, 0);
System.out.println(horarioDeEntrada); //09:00
```

La clase `LocalDateTime` sirve para representar una fecha y hora específicas. Podemos representar una fecha y hora para pruebas importantes o una audiencia judicial.

```
LocalDateTime ahora = LocalDateTime.now();
LocalDateTime aperturaDelMundial = LocalDateTime.of(2020, Month.JUNE, 12, 17,
System.out.println(aperturaDelMundial); //2014-06-12T17:00 (formato ISO-8601)
```

Fechas con zona horaria

Para representar una fecha y hora en una zona horaria específica, debemos usar la clase `ZonedDateTime`.

```
ZoneId zonaHorariaDeSaoPaulo = ZoneId.of("America/Sao_Paulo");
ZonedDateTime ahoraEnSaoPaulo = ZonedDateTime.now(zonaHorarioDeSaoPaulo);
System.out.println(ahoraEnSaoPaulo); //2020-11-08T10:02:57.838-03:00[America/S
```

Con un ZonedDateTime, podemos representar, por ejemplo, la fecha de un vuelo.

```
ZoneId zonaHorariaDeSaoPaulo = ZoneId.of("America/Sao_Paulo");
ZoneId zonaHorariaDeNuevaYork = ZoneId.of("America/New_York");

LocalDateTime salidaDeSaoPauloSinZonaHoraria =
    LocalDateTime.of(2014, Month.APRIL, 4, 22, 30);
LocalDateTime llegadaANuevaYorkSinZonaHoraria =
    LocalDateTime.of(2020, Month.APRIL, 5, 7, 10);

ZonedDateTime salidaDeSaoPauloConZonaHoraria =
    ZonedDateTime.of(salidaDeSaoPauloSinZonaHoraria, zonaHorariaDeSaoPaulo);
System.out.println(salidaDeSaoPauloConZonaHoraria); //2020-11-04T22:30-03:00[A

ZonedDateTime llegadaANuevaYorkConZonaHoraria =
    ZonedDateTime.of(llegadaANuevaYorkSinZonaHoraria, zonaHorariaDeNuevaYork);
System.out.println(llegadaANuevaYorkConZonaHoraria); //2020-11-05T07:10-04:00[

Duration duracionDelVuelo =
    Duration.between(salidaDeSaoPauloConZonaHoraria, llegadaANuevaYorkConZonaH
System.out.println(duracionDelVuelo); //PT9H40M
```

Si calculamos ingenuamente la duración del vuelo, tendríamos 8:40. Sin embargo, como hay una diferencia entre las zonas horarias de São Paulo y Nueva York, la duración correcta es 9:40. Tenga en cuenta que la API ya maneja diferentes zonas horarias.

Otra precaución importante que debemos tomar es en relación al horario de verano. Al final del horario de verano, por ejemplo, ¡el mismo horario existe dos veces!

```
ZoneId zonaHorariaDeSaoPaulo = ZoneId.of("America/Sao_Paulo");

LocalDateTime finDelHorarioDeVerano2013SinZonaHoraria =
    LocalDateTime.of(2020, Month.FEBRUARY, 15, 23, 00);

ZonedDateTime finDelHorarioVerano2013ConZonaHoraria =
    finDelHorarioDeVerano2013SinZonaHoraria.atZone(zonaHorariaDeSaoPaulo);
System.out.println(finDelHorarioVerano2013ConZonaHoraria); //2021-02-15T23:00-

ZonedDateTime UnaHoraMas =
    finDelHorarioVerano2013ConZonaHoraria.plusHours(1);
System.out.println(UnaHoraMas); //2021-02-15T23:00-03:00[America/Sao_Paulo]
```

Observe en el código anterior que, incluso si aumentó en una hora, el tiempo continuó a las 23:00. Sin embargo, tenga en cuenta que la zona horaria ha cambiado de -02: 00 a -03: 00.

Fechas y meses importantes

También existen las clases `MonthDay`, que debe usarse para representar fechas importantes que se repiten cada año, e `YearMonth`, que debe usarse para representar un mes completo para un año específico.

```
MonthDay navidad = MonthDay.of(Month.DECEMBER, 25);
YearMonth Mundial2014 = YearMonth.of(2014, Month.JUNE);
```

Formatear fechas

El `toSting` estándar de las clases de la API utiliza el formato ISO-8601. Si queremos definir el formato de visualización de la fecha, debemos utilizar el método `format`, pasando un `DateTimeFormatter`.

```
LocalDate hoy = LocalDate.now();
DateTimeFormatter formatador =
```

```
  DateTimeFormatter.ofPattern("dd/MM/yyyy");
hoy.format(formatador); //08/11/2020
```

El enum `FormatStyle` tiene algunos formatos predefinidos, que se pueden combinar con un `Locale`.

```
LocalDateTime ahora = LocalDateTime.now();
DateTimeFormatter formatador = DateTimeFormatter
    .ofLocalizedDateTime(FormatStyle.SHORT)
    .withLocale(new Locale("pt", "br"));
ahora.format(formatador); //08/11/20 10:02
```

Manipular fechas

Todas las clases mencionadas tienen diferentes métodos que permiten manipular las medidas de tiempo. Por ejemplo, podemos usar el método `plusDays` de clase `LocalDate` para aumentar un día:

```
LocalDate hoy = LocalDate.now();
LocalDate mañana = hoy.plusDays(1);
```

Otro cálculo interesante es el número de mediciones de tiempo hasta una fecha determinada, que podemos hacer mediante el método `until`. Para averiguar el número de días hasta una fecha, por ejemplo, debemos indicar `ChronoUnit.DAYS` como parámetro.

```
MonthDay navidad = MonthDay.of(Month.DECEMBER, 25);
LocalDate navidadDeEsteAño = navidad.atYear(Year.now().getValue());
long diasAHastaNavidad = LocalDate.now()
    .until(navidadDeEsteAño, ChronoUnit.DAYS);
```

Podemos usar la interfaz `TemporalAdjuster` para definir diferentes formas de manipular las medidas de tiempo. Es interesante notar que esta es una interfaz funcional que permite el uso de lambdas.

La clase auxiliar `TemporalAdjusters` tiene varios métodos que actúan como *factories* para diferentes implementaciones útiles de `TemporalAdjuster`. Podemos, por ejemplo, descubrir cuál es el próximo viernes.

```
TemporalAdjuster ajustadorParaProximoViernes = TemporalAdjusters.next(DayOfWee
LocalDate proximoViernes = LocalDate.now().with(ajustadorParaProximoViernes);
```

Inmutabilidad y Testabilidad

Si agrega un día a un `LocalDate`, la información de la fecha no se modificará.

```
LocalDate hoy = LocalDate.now(); //2020-11-08
hoje.plusDays(1);
System.out.println(hoy); //2020-11-08 (¡aún es hoy, y no mañana!)
```

De hecho, cualquier método que cambie el objeto devuelve una referencia a un nuevo objeto con la información modificada.

```
LocalDate hoy = LocalDate.now();
LocalDate mañana = hoy.plusDays(1);
boolean mismoObjeto = hoy == mañana; //false, ya que es inmutable
```

Esto se aplica a todas las clases del paquete.`java.time`, que son inmutables y, por lo tanto, seguros para subprocesos y más fáciles de mantener.

Otro punto importante de la API es la mejor capacidad de prueba con el uso de la clase `Clock`.

Trabajar con código legado

No podremos cambiar todo nuestro código existente para que funcione con la potencia del paquete `java.time` durante la noche. Por lo tanto, Java 8 trajo algunos puntos de interoperabilidad entre los antiguos `Date` y `Calendar` y la nueva API.

```
Calendar calendar = Calendar.getInstance();
Instant instantAPartirDelCalendar = calendar.toInstant();
```

```
Date dateAPartirDelInstant = Date.from(instantAPartirDelCalendar);
Instant instantAPartirDeLaDate = dateAPartirDelInstant.toInstant();
```

Además, la clase abstracta `Calendar` ganó un *builder*, lo que permite crear una instancia con fluidez.

```
Calendar calendario =
      new Calendar.Builder()
        .setDate(2020, Calendar.APRIL, 8)
        .setTimeOfDay(10, 2, 57)
        .setTimeZone(TimeZone.getTimeZone("America/Sao_Paulo"))
        .setLocale(new Locale("pt", "br"))
        .build();
```

La nueva API de fecha de Java 8 es bastante extensa, con varias otras características interesantes. Ciertamente, es una adición muy bienvenida a Java, que facilita enormemente el trabajo de manipulación de fechas.

En Alura tenemos toda una formación con varios cursos de **Java**, desde lo más básico con el lenguaje hasta conceptos más avanzados como clases, poliformismo y herencia. Pero no te asustes con esos términos, ellos son abordados con un proyecto práctico que junto con nuestra metodología y didáctica hacen que el contenido sea mucho más fácil de ser aprendido.

ARTÍCULOS DE TECNOLOGÍA > PROGRAMACIÓN

ALIADOS

Empresa participante do SCALE UP ENDEAVOR

En Alura somos unas de las Scale-Ups seleccionadas por Endeavor, programa de aceleración de las empresas que más crecen en el país.

Google for Startups Alumni 2021

Fuimos unas de las 7 startups seleccionadas por Google For Startups en participar del programa Growth Academy en 2021

POWERED BY

CURSOS

Cursos de Programación
Lógica de Programación | Java

Cursos de Front End
HTML y CSS | JavaScript | React

Cursos de Data Science
Data Science | Machine Learning | Excel | Base de Datos | Data Visualization | Estadística

Cursos de DevOps
Docker | Linux

Cursos de Innovación y Gestión